Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Towards Efficient Comparison of Change-Based Models

Alfa Yohannis[ac] Horacio Hoyos Rodriguez[a] Fiona Polack[b]
Dimitris Kolovos[a]

a. Department of Computer Science, University of York, United Kingdom
b. School of Computing and Maths, Keele University, United Kingdom
c. Department of Computer Science, Kalbis Institute, Indonesia

Abstract Comparison of large models can be time-consuming since every element has to be visited, matched, and compared with its respective element in other models. This can result in bottlenecks in collaborative modelling environments, where identifying differences between two versions of a model is desirable. Reducing the comparison process to only the elements that have been modified since a previous known state (e.g., previous version) could significantly reduce the time required for large model comparison. This paper presents how change-based persistence can be used to localise the comparison of models so that only elements affected by recent changes are compared and to substantially reduce comparison and differencing time (up to 90% in some experiments) compared to state-based model comparison.

Keywords Model Comparison; Change-based Persistence; State-based Persistence; Partial Model.

1 Introduction

In modelling and model management, it is common to find that many versions or variants of a model exist. These versions are commonly persisted as snapshots of the model at a given point in time, in a state-based format such as XMI. Model comparison activities can be applied to the different versions of a model to highlight their differences: changes in properties values, new elements, etc. However, comparing versions of large file-based[1] models in a state-based format can be computationally expensive since both versions of the model need to be loaded in memory in their entirety before their elements can be matched and diffed.

[1]Persisting models in databases involves its own challenges which have been discussed extensively in the literature. For the rest of the paper, we are only concerned with file-based models and we return to database-backed model representations in Section 6.

Alfa Yohannis, Horacio Hoyos Rodriguez, Fiona Polack, Dimitris Kolovos. *Towards Efficient Comparison of Change-Based Models*. In Journal of Object Technology, vol. 18, no. 2, 2019, pages N:1–20.
doi:10.5381/jot.2019.18.2.aN

In our previous work [YKP17, YRPK18a, YRPK18b], we proposed change-based persistence (CBP) as an alternative approach to state-based persistence of EMF models [SBMP08]. Instead of persisting models as XMI snapshots, in the proposed approach models are persisted as a complete history of changes. We demonstrated the substantial performance benefits of CBP in terms of saving changes to large models [YKP17] as well as the method for reducing model loading time compared to naively replaying all recorded change events [YRPK18b] to reconstruct the state of a change-based model. In this paper, we demonstrate how a change-based representation also enables much more efficient and performant model comparison between versions of the same model. Our experiments, presented in Section 5, demonstrate savings of the order of 90% for (relatively) small changes made to large models.

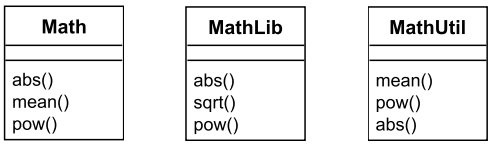
This paper is structured as follows. Section 2 provides an overview of our previous work on change-based model persistence. Section 3 discusses state-based model comparison. Section 4 presents our change-based approach to speed up model comparison and its implementation. Section 5 reports on the results of evaluation experiments used to evaluate the proposed approach. Section 6 provides an overview of related work, and Section 7 concludes with a discussion on directions for future work.

2 Change-based Persistence

CBP is an alternative approach to state-based persistence (SBP) of models. Instead of persisting snapshots of the state of a model – which is the default behaviour of frameworks such as EMF – CBP persists the entire history of change events of a model [YRPK18a]. For example, in the SBP approach, when we save the UML class diagram in Fig. 1a in standard XMI format, we only record the last state of the model, as shown in List. 1. In contrast, when we develop the same model in the CBP approach, all the events generated from modifying the model are captured and persisted in the model file as shown in List. 2[2]. Each change event contains information about the type of the operation applied as well as the as values, elements, or features involved. Replaying the change events in List. 2 produces the same eventual model as in Fig. 1a.

```
1 <uml:Class id="x" name="Math">
2 <operation id="a" name="abs"/>
3 <operation id="b" name="mean"/>
4 <operation id="c" name="pow"/>
5 </uml:Class>
```

Listing 1 – The simplified XMI of the model in Fig. 1a.



(a) origin (b) left (c) right

Figure 1 – Different versions of a model.

```
1 create x type Class
2 set x.name to "Math"
3 create a type Operation
4 set a.name to "abs"
5 create b type Operation
6 set b.name to "mean"
7 create c type Operation
8 set c.name to "pow"
9 add a to x.operations at 0
10 add b to x.operations at 1
11 add c to x.operations at 2
```

Listing 2 – The pseudo-formatted CBP of the model in Fig. 1a.

[2] In our implementation, the change-based format is XML-based.

3 State-based Model Comparison

In a collaborative modelling setting, a model can have different versions. Consider the case where an initial version of a model exists in a Version Control System (VCS) server (Fig. 2). Two modellers, Bob and Alice, check out the original model (steps 1 and 2) to their local machines and modify it (steps 3 and 4). Alice then commits her work (original + Alice's changes) to the VCS. Since there is no newer commit on the VCS, the commit process is straightforward (step 5). Bob then decides to also commit his work (original + Bob's changes) to the VCS. However, he needs to merge his work with the current updated version at the VCS since his last checkout. His machine downloads the latest version from the server (step 6), i.e. Alice's version. To merge his and Alice's changes, Bob needs to perform model comparison to check their differences, resolve possible conflicts between the models, and then merge them (step 7). After that, he can push it back to the VCS server.

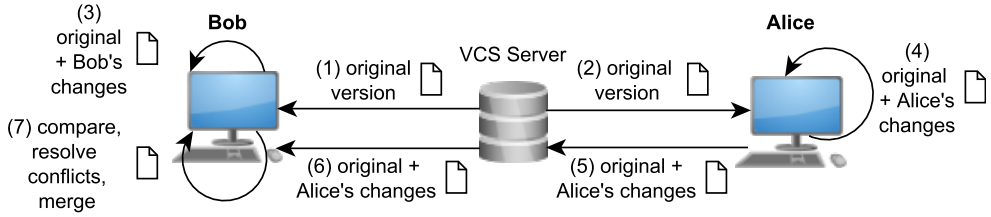


Figure 2 – A usecase of CBP in a collaborative modelling.

In a SBP setting, Bob produces the model in Fig. 1b (the left model), and Alice the model in Fig. 1c (the right model) producing XMI files as shown in List. 3 and List. 4 respectively. Before Bob can merge, he must compare the right model with the left model. In state-based comparison, comparing models commonly consists of two steps: *matching* and *diffing*. The matching process establishes matches between the elements of both models, to determine the elements in the left model that correspond to elements in the right model. Generally, the matching process iterates through all the elements of the models being compared and matches them by their identifiers or through a similarity mechanism [BKL+12, EMF].

The diffing process identifies differences between the matched elements [BKL+12, EMF].Differences between the matched elements and all their features is usually done using a Longest Common Subsequence (LCS) algorithm, e.g., [Mye86].

```
1  <uml:Class id="x" name="MathLib">
2    <operation id="a" name="abs/>
3    <operation id="d" name="sqrt"/>
4    <operation id="c" name="pow"/>
5  </uml:Class>
```

Listing 3 – The simplified XMI of the left model in Fig. 1b.

```
1  <uml:Class id="x" name="MathUtil">
2    <operation id="b" name="mean"/>
3    <operation id="c" name="pow"/>
4    <operation id="a" name="abs"/>
5  </uml:Class>
```

Listing 4 – The simplified XMI of the right model in Fig. 1c.

In our example, the matching process in state-based comparison – as performed by EMF Compare [EMF] – iterates through all the elements of both models and matches them using their identifiers. The matching process yields 3 matches: m_1 = (x, x), m_2 = (a, a), and m_3 = (c, c), and 2 unmatched elements, um_1 = (d, -) and um_2 = (-, b).

The diffing process then iterates through all the matches and unmatched elements and uses an LCS algorithm to identify their differences. In the first match, it identifies

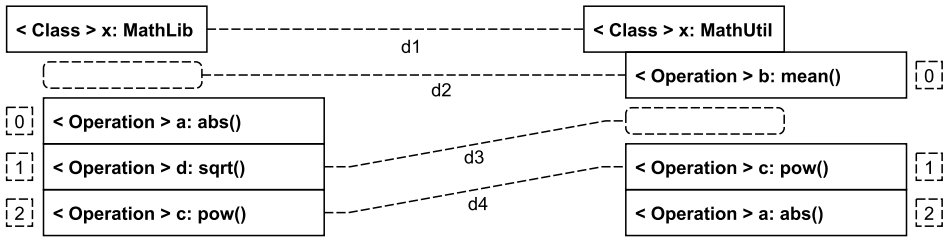


Figure 3 – A model comparison of the left and right models in Listings 3 and 4.

that the elements x are different in their name and operations features. The left x's name is "MathLib" while the other x's name is "MathUtil" (diff ds_1). The operations features are different in their contents – the left operations feature does not contain element b (diff ds_2), the left operations feature contains element d that does not exist in the right operations (diff ds_3), and the indexes of element c are different in both features (diff ds_4). It is important to note that the employed LCS algorithm does not detect the different position of element A as a difference; it only identifies the minimum number of differences which if all are resolved unidirectionally can make both models equal. Otherwise, the number becomes less optimal – not minimum.

Differences are commonly expressed as a list of changes that must be applied to a target model so that it is made equal to a reference model. This paper treats the left model as a reference model and the right model as the target model. This means that differences are expressed as changes applied to the right model so that it equals the left model. To express differences, we use the following terms: LeftContainer, RightContainer, LeftFeature, RightFeature, LeftIndex, RightIndex, LeftValue, RightValue, and Kind. The *Container, *Feature, and *Value are the target element, feature, and value involved in a difference (* symbol can be replaced with Left and Right). *Index is the index of a value in a feature. Kind is the type of difference. It can be one of these types: CHANGE, ADD, DELETE, and MOVE. CHANGE means a pair of single-valued features have different values. ADD indicates that a value does not exist in the right model, thus it requires the addition of the value. DELETE is the opposite of ADD. MOVE indicates that matched elements differ in terms of their containers, containing features, or indexes. A Container is an element that contains a value. A containing feature is a feature owned by a container in which a value is contained. An index is the position of a value in a containing feature.

Based on these definitions, we can express the result of the diffing process as: $ds_n = [LeftContainer_n, RightContainer_n, LeftFeature_n, RightFeature_n, LeftIndex_n, RightIndex_n, LeftValue_n, RightValue_n, Kind_n]$. Thus, ds_1 = [x, x, name, name, 0, 0, "MathLib", "Mathutil", CHANGE], ds_2 = [x, x, operations, operations, null, 0, null, b, DELETE], ds_3 = [x, x, operations, operations, 1, null, d, null, ADD], and ds_4 = [x, x, operations, operations, 2, 1, c, c, MOVE]. We can use this information to represent the differences visually as depicted in Fig. 3. Applying these differences as changes to the right model will transform it into the left model.

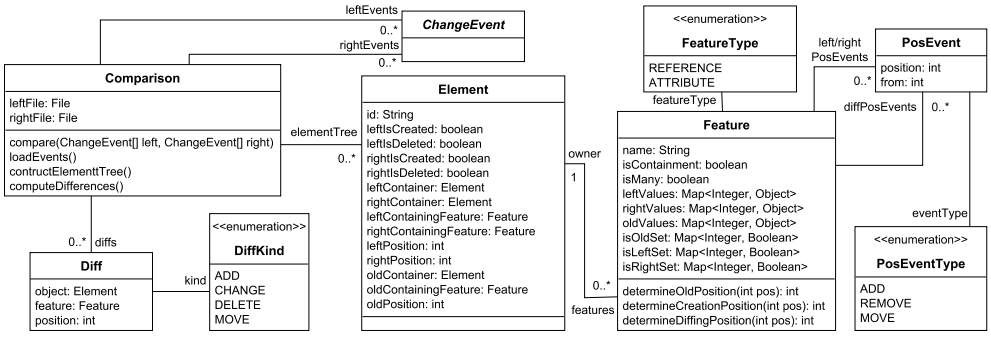


Figure 4 – A class diagram showing the core components of the change-based approach to speed up model comparison.

4 Change-based Approach for Comparing Models

Now let's consider the same example in a CBP setting. The changes made by Bob and Alice are appended to their local original CBP producing two different CBP representations as displayed in Listings 5 and 6[3] – capturing different courses of modification made by the two modellers. Then, the example is the same with Alice committing her changes and Bob wanting to merge Alice's work with his.

```
12  set x.name from "Math" to "MathLib"
13  create d type Operation
14  set d.name to "sqrt"
15  add d to x.operations at 1
16  remove b in x.operations at 2
17  delete b
```

Listing 5 – The appended changes made by Bob to produce the model in Fig. 1b (left version).

```
12  move a in x.operations from 0 to 2
13  set x.name from "Math" to "MathUtil"
```

Listing 6 – The appended changes made by Alice to produce the model in Fig. 1c (right version).

In CBP, comparison has three phases: event loading, element tree construction, and diff computation. Further, comparison is not performed over all the elements of the model; instead, we only need to compare the last set of changes from the source and reference model. The last set of changes can be identified easily by finding their last common change. A simplified class diagram of our approach's implementation[4] is depicted in Fig. 4. Next, we describe the three phases in detail.

4.1 Event Loading

In the event loading phase, our implementation loads change events recorded in two CBP files into memory. The most important aspect of this phase is the partial loading as only lines starting from the position where the two files are different are loaded. Thus, not the whole model needs to be traversed and loaded. In this case, lines 1-11 in List. 2 are skipped.

[3]Both CBPs only present the changes after the last line of the original version (start from line 12).
[4]The source can be found at https://github.com/epsilonlabs/emf-cbp.

Only lines starting from line 12 in Listings 5 and 6 are loaded, yielding two partial – left and right – change-event models.

4.2 Element Tree

An element tree is a representation of the changes of model elements in the source and reference models. It contains detailed information about elements and their properties. It contains similar information to that captured in change lists in SBP, but also provides more information about the changes. For example, the element tree can keep track of a feature's old value and element/value's indexes inside multi-valued properties. The element tree only contains the partial states of affected elements of the original, left, and right models as depicted in Figures 5 and 6.

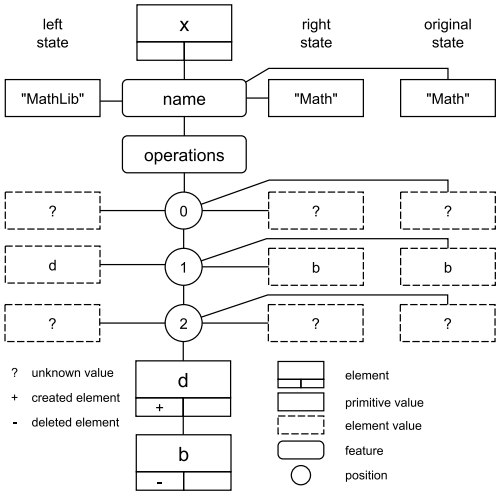


Figure 5 – The elementTree after processing all left change events.

To better understand the construction of an element tree from change events, we use the following running example using both change events in the Listings 5 and 6. We start from the left change events.

4.2.1 Left Side

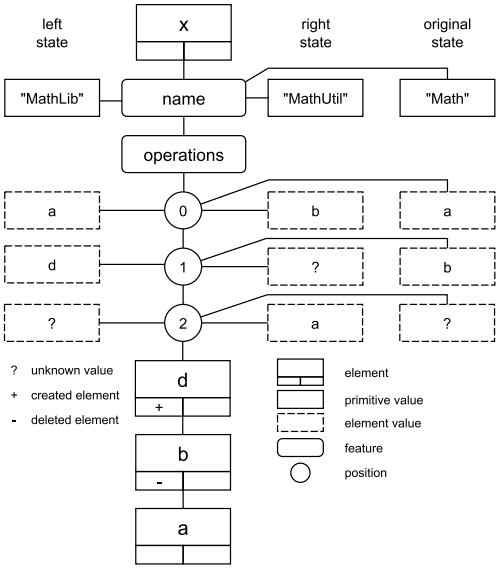


Figure 6 – The elementTree after processing all left and right change events.

From the first event [**set** `x.name` **from** `"Math"` **to** `"MathLib"`] at line 12, we can identify that an element with id x has existed from the original model. It has a feature name with a value "Math" in the original model that has been changed to "MathLib" in the left model. Since the element x does not already exist in the elementTree, we create its instance of Element and also its feature name. We set the value of the feature name to "MathLib" and also set it to "Math" in the partial state of the original model – it has not been set before. As this feature on the right side also has not been set, we set it to "Math" as well.

At line 13, in the event [**create** `d` **type** `Operation`], we can identify that an element with id d has been created. We also update the elementTree to include this element and set the element's flag leftIsCreated to true. In the event [**set** `d.name` **to** `"sqrt"`] at

line 14, we can identify that element d's feature name has been set to "sqrt". Thus, we update d's feature name in the elementTree. From the event [**add** `d` **to** `x.operations` **at** `1`] at line 15, we can deduce that element d is added to index 1 in the element x's feature operations. Thus, we assign d to element x's feature operations at index 1 in the elementTree. As d is a new element that only exists in the left model, we do not update changes of this element to the original and right models.

From the event [**remove** `b` **in** `x.operations` **at** `2`] at line 16, we can identify that there is element b in the original model, but it is deleted in the left model. The index of element b in the original model can be calculated back through the previous change events that have been applied to its feature. Since the previous event is adding element d to index 1 and the index of b is at 2 at the time it is removed, we can deduce that before element d is added, the index of element b is at 1 and is shifted to 2 because of the addition of element d. Therefore, we can conclude that the original index of element b is at 1. Thus, we update the original state of the elementTree by adding element b into the element x's feature operations at index 1.

We perform the same procedure to also add element b to the right state of the elementTree. However, since no change event has been applied to the right side of element x's feature operations, the calculation of element b's index should return the same value as in the original state (line 13, Alg. 1), and thus element b has the same index as in the original state. It is important to notice, in this step, the flag isRightSet (class Feature, Fig. 4) is not set to true since we want the value to be able to be overridden during processing of the right change events. The last event [**delete** `b`], removes the element b from the left model. Hence, we set the flag leftIsDeleted of element a to true.

Fig. 5 illustrates the state of the elementTree after all left change events have been processed. As can be seen, the elementTree exhibits the partial states of the original, left, and right models at once.

4.2.2 Right Side

From the first event [**move** `a` **in** `x.operations` **from** `0` **to** `2`] at line 12, we can infer that in the right model there is an element with id a positioned at index 2 in the element x's feature operations. Thus, element a – an instance of class Element in 4 – is added to the elementTree and positioned at index 2 of the element x's feature operations. Since the event is a move type and the new index is larger than its previous index, elements that are between its previous and new indexes are shifted one place down. As element b has already existed in the same feature (the element was added during the process of the left change events) and its index is between element a's movement, the index of element b is shifted down from 1 to 0.

Also, since the event's type is move and its previous index is 0 and it is the first event that changes the index of element a, these conditions imply that element a in the original model is positioned at index 0 in the element x's feature operations. Therefore, we add the element a to element x's feature operations in the original state of the elementTree. Since the index 0 in the element x's feature operations has not been set, we also add element a to that index in the right state of the elementTree. From the last event [**set** `a.name` **from** `"Math"` **to** `"MathUtil"`] at line 13, we can infer that in the right model, the value of element a's feature name is "MathUtil". Hence, we set

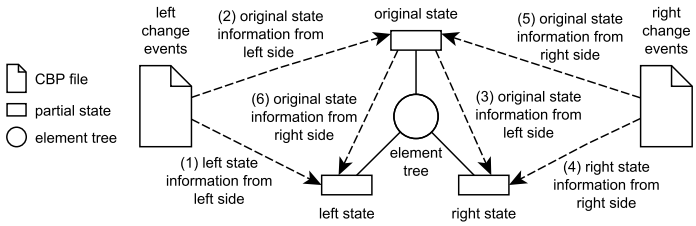


Figure 7 – Steps in Element Tree construction.

the feature name to "MathUtil" in the right state. We do not apply this operation to the original and left states as they have been set before. Fig. 6 exhibits the state of the elementTree after both sides' change events have been processed.

The construction of the elementTree that we have just explained follows the steps shown in Fig. 7. First, the partial state S_L of the left model in the elementTree is constructed based on the information retrieved from the left change events (step 1). We denote this information as I_{LL}. We can also construct the partial state S_O of the original model using the information related to the original state contained in the left change events I_{OL} (step 2). The information I_{OL} allows us to construct the initial partial state S_R of the right model (step 3). Similarly, using the information from the right change events I_{RR}, we update the partial right state S_R that has been initialised before using the information I_{OL} (step 4), implying that $I_{OL} \cup I_{RR} \rightarrow S_R$. Also, information related to the state of the original model from the right change events I_{OR} is used to update the original state (step 5). Thus, we have a partial state of the original model constructed using information from both left and right sides, $I_{OL} \cup I_{OR} \rightarrow S_O$. Finally, we also use the information I_{OR} to update the partial state of the left model (step 6), implying that $I_{LL} \cup I_{OR} \rightarrow S_L$.

Alg. 1 describes the steps presented in Fig. 7 in a generic fashion. It iterates through all of a model's change events and uses the information contained in them to construct the relevant partial state. The selection of side, left or right change events, that are executed first depends on the Side enumeration value – left or right – passed through the parameter side (the second input parameter). In our implementation, we process the left side first by default. The algorithm also receives an input of the change events events that are to be iterated and the element tree elementTree that has been instantiated before, and then returns the elementTree as output after updating it.

For each event in the events, we collect information needed to build up the elementTree (lines 3-9), such as targetElement, feature, value, previousValue, index, and previousIndex. The targetElement is the element modified by a change event (e.g., x and d in List. 5). This targetElement – an instance of class Element in Fig. 4 – is retrieved from the elementTree if it already exists. Otherwise, a new element is created and added to the elementTree (line 3). In this step we also set the flags *IsCreated and *IsDeleted of the element in Fig. 4. For example, if the type of the event is create then *IsCreated is set to true. The feature – an instance of class Feature in Fig. 4 – represents the target element's feature (e.g., name and operations in List. 5) modified by a change event. It is retrieved from the targetElement's feature list, and a new one is created and added to the targetElement's feature list if the feature does exist (line 5).

The value is the value assigned to the feature in a change event (line 5, Alg. 1). The

value can be the type of Element (e.g., elements b and d, lines 17-18, List. 5) or primitive (e.g., the string "MathLib" at line 14 in the List. 5). The previousValue represents the previous value of the modified feature (line 6, Alg. 1). The previousValue is not defined if no previous value has been assigned. For value and previousValue with type Element, the elements that they represent are retrieved from the elementTree, and if they do not exist, new instances are created. If the type is primitive, the value is treated as it is. Not every change event has a value, particularly events with type create or delete which only modify a target element not the element's feature.

Algorithm 1: Algorithm to construct an element tree from events.

```
   input  : a list of ChangeEvent events
   input  : an enumeration of Side side
   input  : an instance of ElementTree elementTree
   output : an instance of ElementTree elementTree
 1 begin
 2     foreach event in events do
 3         targetElement ← getOrCreateNewTargetElement(event, elementTree);
 4         feature ← getOrCreateNewFeature(event, targetElement);
 5         value ← getValue(event);
 6         previousValue ← getPreviousValue(event);
 7         index ← getIndex(event);
 8         previousIndex ← getPreviousIndex(event);
 9         featureEventList ← getFeatureEventList(feature, side);
           // put all values to their proper indexes
10         updateTree(targetElement, feature, value, index, side);
11         oldIndexes ← calculateOldIndex(featureEventList, previousIndex, side);
12         if not isCreated(value, side) and not isOldValueSet(feature, previousValue,
            previousIndex, side) then
13             setOldValue(feature, previousValue, oldIndex, side);
14             oppositeFeatureEventList ← getOppositeFeatureEventList(feature, side);
15             oppositeIndex ← calculateOppositeIndex(oppositeFeatureEventList, oldIndex,
                side);
16             if not isDeleted(value, side) and not isOppositeSideValueSet(feature, value,
                oppositeIndex, side) then
17                 setOppositeSideValue(feature, value, oppositeIndex, side);
18             end
19         end
20         addEventToFeatureEventList(event, featureEventList);
21     end
22     return elementTree;
23 end
```

The index is the index assigned by a change event to a value in a feature, while previousIndex is the previous index of the value (lines 7-8, Alg. 1). In one change event, we can get both index and previousIndex or only one of them depending on the type of the change event. For example, we can only obtain that the index of d is 1 (line 17 in List. 5) as the change event type is add. In a remove change event, we can only get the previousIndex of b, that is 2 (line 17 in List. 5), as the element does not exist anymore in the left model. We can obtain both of them only in a move change event as an element is moved from a previous index to a new one (line 14 in List. 6). For a single-valued feature, the index and previousIndex are always 0 as the feature can only contain a single value.

At line 9, we retrieve the featureEventList from the feature to be added later with the current event (line 19). The featureEventList is a list – a history – of change events that have been processed that are specific to the feature on the selected side. Using the

obtained targetElement, feature, value, and index, the process then updates the state of the elementTree on the selected side (line 10). After that, it calculates back the original index of a value using the featureEventList and previousIndex (line 11). If the value at oldIndex in the feature has not been set, then the algorithm sets the feature with the previousValue at the oldIndex in the partial state of the original model (lines 12-13). At lines 14-18, the algorithm also does the same thing to the opposite side – if the current side is left then it is right.

4.3 Diff Computation

Using the elementTree presented in Fig. 6, we can determine the difference between the left and right models without having to compare all their elements and features. After the elementTree has been constructed, we iterate through elements and features of the elementTree and use the flags, containers, containing features, and indexes on both sides of each element and value to identify differences between both left and right models. We follow the steps in Alg. 2. The algorithm visits each element and every index of each feature (lines 3-5). At every index, it retrieves the leftValue and rightValue (lines 5-7), passing these, together with the element, feature, and index to a function identifyDiffUsingRules (line 8). The function identifies differences using a set of pre-defined rules which determines differences diffs based on the states of flags of an element, flags and attributes of the element's feature, values of the feature, and indexes of the values. The obtained diffs are then added to the overall list of differences diffList which is output (line 8-9, 13).

Algorithm 2: Algorithm to determine differences.

```
   input  : an instance of ElementTree elementTree
1  begin
2      diffList ← DiffList();
3      foreach element in elementTree do
4          foreach feature in getFeatures(element) do
5              foreach index in getIndexes(feature) do
6                  leftValue ← getLeftValue(feature, index);
7                  rightValue ← getRightValue(feature, index);
                   // rules starts from here
8                  diffs ← identifyDiffUsingRules(element, feature, leftValue, rightValue,
                     index);
9                  addToDiffList(diffs,diffList);
10             end
11         end
12     end
13     return diffList;
14 end
```

We illustrate the principles and use of rules by discussing the rules used to identify differences in the running example, which can be found in Alg. 3. The algorithm is the breakdown of the function identifyDiffUsingRules in Alg. 2. As previously stated, it is important to remember that we use the left model as a reference which means the differences are presented as changes that transform the right model to become equal to the left model.

The first rule (Rule 1) in Alg. 3 is to identify changes in single-valued attributes. A feature has to be of type attribute, both side values have to be different, and the

element should have not been created or deleted in both models. The second rule (Rule 2) identifies whether an element is in a different location in both models. The element must not have been deleted and must exist from the previous version – the original model. Also, its containers, containing features, or indexes of the element have to be different on both sides.

Algorithm 3: Some rules to determine differences.

```
   input : an Element element, a Feature feature, a variable leftValue, a variable rightValue,
           an Integer index
   output: a List of Diff diffs
 1 diffs ← createDiffList();
   // ...
   // Rule 1:  a rule to determine a change of a single-valued attribute
 2 if getType(feature) is Attribute and isSingleValued(feature) and leftValue <> rightValue
     and not leftIsCreated(element) and not leftIsDeleted(element) and not
     rightIsCreated(element) and not rightIsDeleted(element) then
 3 |   diff ← createNewDiff(element, element, feature, feature, index, index, leftValue,
   |     rightValue, DifferenceType.CHANGE);
 4 |   addDiffToDiffList(diff, diffs);
 5 end
   // Rule 2:  one of rules to determine movement of an element
 6 if getType(feature) is Containment and not leftIsCreated(leftValue) and not
     leftIsDeleted(leftValue) and not rightIsCreated(leftValue) and not
     rightIsDeleted(leftValue) and (getLeftContainer(leftValue) <>
     getRightContainer(leftValue) or getLeftFeature(leftValue) <> getRightFeature(leftValue)
     or getLeftIndex(leftValue) <> getRightIndex(leftValue)) then
 7 |   diff ← createNewDiff(getLeftContainer(leftValue), getRightContainer(leftValue),
   |     getLeftFeature(leftValue), getRightFeature(leftValue), getLeftIndex(leftValue),
   |     getRightIndex(leftValue), leftValue, leftValue, DifferenceType.MOVE);
 8 |   addDiffToDiffList(diff, diffs);
 9 end
   // Rule 3:  one of rules to determine deletion of an element
10 if getType(feature) is Containment and not leftIsCreated(rightValue) and
     leftIsDeleted(rightValue) and not rightIsCreated(rightValue) and not
     rightIsDeleted(rightValue) then
11 |   createNewDiff(getLeftContainer(rightValue), getRightContainer(rightValue),
   |     getLeftFeature(rightValue), getRightFeature(rightValue), getLeftIndex(rightValue),
   |     getRightIndex(), rightValue, null, DifferenceType.DELETE);
12 |   addDiffToDiffList(diff, diffs);
13 end
   // Rule 4:  one of rules to determine addition of an element
14 if getType(feature) is Containment and leftIsCreated(leftValue) and not
     leftIsDeleted(leftValue) and not rightIsCreated(leftValue) and not
     rightIsDeleted(leftValue) then
15 |   diff ← createNewDiff(getLeftContainer(leftValue), getRightContainer(leftValue),
   |     getLeftFeature(leftValue), getRightFeature(leftValue), getLeftIndex(leftValue),
   |     getRightIndex(leftValue), null, rightValue, DifferenceType.ADD);
16 |   addDiffToDiffList(diff, diffs);
17 end
   // ...
18 return diffs
```

The third rule (Rule 3) identifies the deletion of an element. If an element in the left model is not created but exists in the model, it means that the element has been there from the previous version – the original model. This also means that the element also exists in the right model, unless it has been deleted. Thus, in order to make the right model equal to the left model, the element has to be deleted also in the right model. The fourth rule (Rule 4) identifies the need for an addition of an element. If

an element is created in the left model and has not been deleted, it means that the element should be added also to the right model to make both models equal.

In the running example, when the iteration of the elementTree (Fig. 6) returns feature name, the type of the feature is a single-valued attribute and both sides of the feature are different in their values, this means that the condition of the first rule is met. Thus, we can conclude that in order to make the left value of the feature equal to the right value, we must override the value "MathUtil" with "MathLib"; the type of this difference is CHANGE. When the iteration is at index 0 in the element x's feature operations, we have two values: the leftValue is element a, and the rightValue is element b. As a exists on both sides – all *Created and *Deleted flags are false, and it also has a different index, at 0 in the left state and 2 in the right state. This meets the condition of the second rule. Thus, we can conclude that in order to make the index of element a in the right model equal its index in the left model, element a should be moved from index 2 to 0. Thus, the type of this difference is MOVE.

Element b used to exist but has been deleted from the left model (flags leftIsCreated = false, leftIsDeleted = true); it still exists in the right state (flags rightIsCreated = false, rightIsDeleted = false). This condition satisfies the third rule. Therefore, the element b should be deleted from the right model; the type of this difference is DELETE. We can get only one value when the iteration is at index 1 in the element x's feature operations; the leftValue is element d, but the rightValue is unidentified. Thus, we only process the leftValue. Element d is only created in the left model (flags leftIsCreated = true, leftIsDeleted = false, rightIsCreated = false, rightIsDeleted = false). This meets the condition of the fourth rule. Thus, to make element d also exist in the right state, we must add it into element x's feature operations at index 1. Therefore, the type of this difference is ADD. At index 2, the element a is skipped because it has been processed already.

Similar to the state-based approach in Section 3, we express identified differences as dc_n = [$LeftContainer_n$, $RightContainer_n$, $LeftFeature_n$, $RightFeature_n$, $LeftIndex_n$, $RightIndex_n$, $LeftValue_n$, $RightValue_n$, $Kind_n$]. Thus, dc_1 = [x, x, name, name, 0, 0, "MathLib", "Mathutil", CHANGE], dc_2 = [x, x, operations, operations, ?, 0, ?, b, DELETE], dc_3 = [x, x, operations, operations, 1, ?, d, ?, ADD], and dc_4 = [x, x, operations, operations, 0, 2, a, a, MOVE]. This change-based approach might produce differences that are distinct from differences identified using state-based approach. This can be seen between by comparing ds_4 and dc_4 ($ds_4 \neq dc_4$, [x, x, operations, operations, 2, 1, c, c, MOVE] ≠ [x, x, operations, operations, 0, 2, a, a, MOVE]). In the state-based approach, element c has a MOVE difference – it has different index (ds_4), while in the change-based approach, this difference is attributed to element a (dc_4). However, in both approaches, if we resolve their differences by performing all-left-to-right merging – making the right model equal to the left model, both approaches produce two models that are equivalent. In this way, we can check the correctness of the identified differences produced by the change-based approach.

5 Evaluation

In this section, we present the method that we employed to evaluate our change-based comparison approach and discuss the results. We also present the limitations and threats to the validity of the evaluation.

5.1 Method

In order to assess the performance benefits of the change-based approach in terms of model comparison, we have evaluated it against a mature and widely-used state-based comparison tool (EMF Compare [EMF, Eclb]). Since there are no manually developed, large models persisted in our change-based format yet, the dataset for our experiments was constructed from a large model reverse-engineered from the Eclipse Epsilon project [Ecld, Eclc]. This model conforms to the Java metamodel [Ecle] and consists of more than 1.6 million elements with a size of 224 MBs when persisted in XMI.

We cloned the original model to produce two new (left and right) models and perform operations (add, remove, move, set with random elements, features, indexes, and values) on both models to create differences. We made 1.1 million artificial changes to each model, generating over 1.1 million events (one operation can generate more than one event, e.g., a move between features generates remove and add events). Events generated by the changes were persisted in our change-based format (to be used later in change-based model comparison). After every 50,000 changes, we made a measurement point. We persisted the last state of the models in state-based format (to be used later in state-based model comparison) and then performed change-based and state-based model comparison and measured their execution time and memory footprint. We created 22 measurement points to capture their trends in one experiment.

We conducted five experiments. In the first experiment, the ratio of occurrence between add, remove, move, and set changes is set to 1:1:20:40 intuitively in assumption that in a mature model modification – move and set events – occurs more frequent than addition and deletion. Since we wanted the change of total elements not to affect our measurement, the number of total elements should be kept constant. For example, it is difficult to tell an increase of time in comparison is caused by an increase in the number of elements or by the number of change events. One way to do this was to exclude add and remove operations. However, excluding both operations made measurement less representative. Thus, we still included both operations but made their probabilities equal so that the number of total elements remain largely unchanged. In the rest of the experiments, we only performed homogeneous type operations – isolated from other types – per experiment (e.g., add-only, move-only operations). In the end, we obtained 5 results of the experiments: mixed, add-only, remove-only, move-only, and set-only measurement results. We did this to asses whether operations of different types have a different impact on model comparison.

For the change-based approach, the comparison time comprises loading change events, constructing an element tree, and identifying differences. The memory footprint is the space used to hold the change events, element tree, and differences in memory. For the state-based approach, the comparison time comprises matching elements and identifying differences, and the memory footprint is the space required to hold the matches and differences in memory. All measurements were performed on the same machine with the following specification: AMD Opteron(tm) Processor 6386 SE @ 2.8 GHz cache size 2 GBs (64 processors), 528 GBs main memory, Ubuntu 16.04.6 LTS operating system, and Java(TM) SE Runtime Environment (build 1.8.0_201-b09) with JVM InitialHeapSize 2GBs and MaxHeapSize 32 GBs.

Since the change-based and state-based approaches can produce a different number of

syntactically equivalent differences, in order to evaluate the correctness of the change-based approach, we reconciled all the differences by performing all-left-to-right merging – making the right model identical to the left model – based on the identified differences. If the all-left-to-right merging of change-based approach produces a model that is identical to the model produced by the all-left-to-right merging of the state-based approach then it can be said that differences identified by the change-based approach are correct. We performed this correctness checking at every measurement point.

5.2 Results and Discussion

In this section, we report on the obtained results in terms of comparison time and memory footprint for the mixed and homogeneous operation experiments.

5.2.1 Mixed Operations

In the mixed operation measurement, we modify two identical models differently by applying random operations. As the number of change events generated by the modification grows, the numbers of affected elements and differences also increase in a logarithmic manner. The patterns can be seen in Fig. 8. The growth is logarithmic since the probability that the random operations modify the same elements also increases. Thus, some change events might not contribute to the addition of new affected elements and differences. In other words, more events are required to increase the number of affected elements or differences. In Fig. 8, the total number of elements remains largely unchanged due to the equal probabilities of addition and deletion as has been set in Section 5. The figure gives us an insight about the characteristics of the modification caused by the random operations in the mixed operation measurement; it supports explaining the implication of the changes on execution time and memory footprints of model comparison.

After applying some random changes on both models, the modification produces 100,000 change events at the first measurement point. Using this amount of

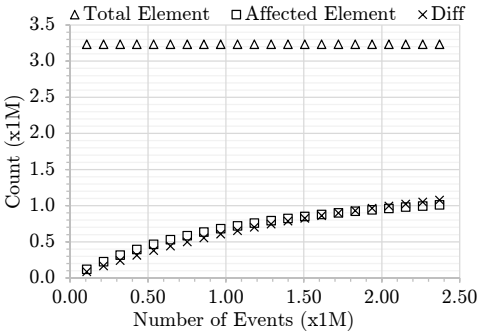
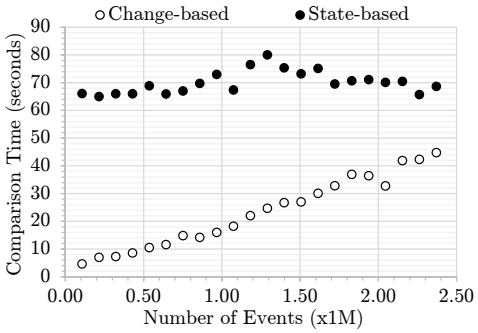
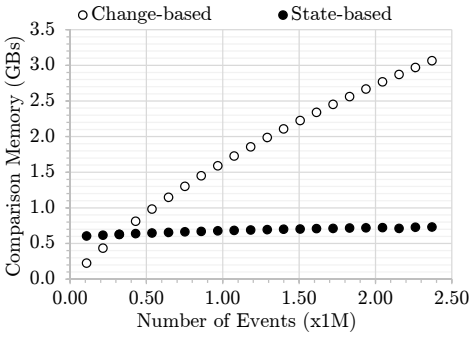


Figure 8 – total elements, affected elements, and diffs



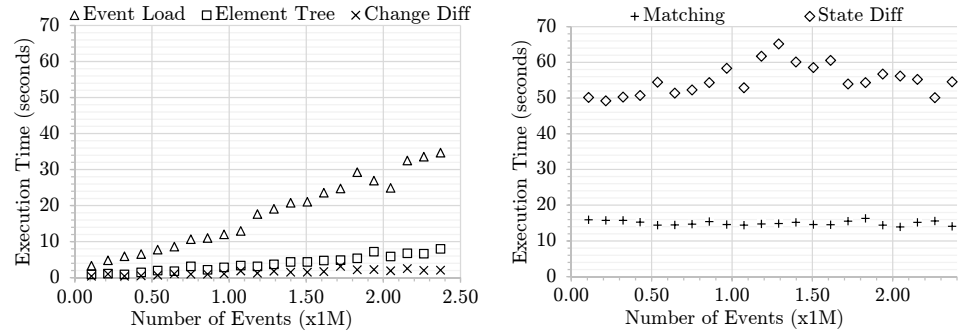
(a) execution time



(b) memory footprint

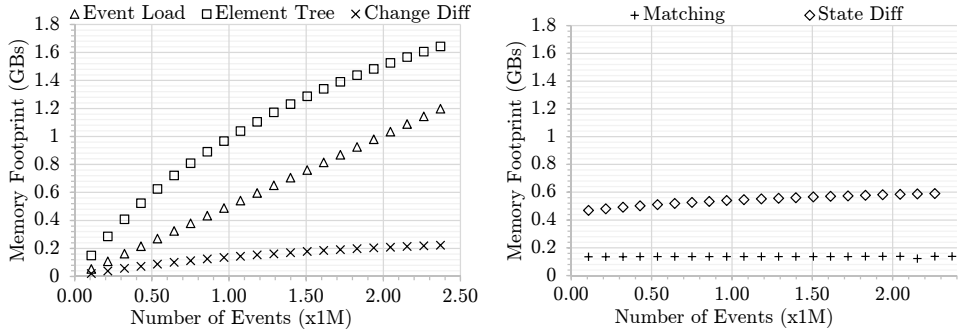
Figure 9 – Change-based vs. state-based model comparison as differences increase.

events, our change-based comparison only takes 5 seconds to identify around 90,000 differences, in contrast to the state-based comparison that takes 66 seconds (see the first measurement points in Figures 8 and 9a). If the modification continues, more change events are generated. This growing number of change events has to be loaded into memory and thus slows down the change-based comparison. Nevertheless, the change-based comparison is still faster than the state-based comparison even though the number of change events reaches 2.37 million – more than 1 million differences at that point; the change-based comparison outperforms the state-based comparison in execution time (Figure 9a). Fig. 10a breaks down the comparison time in detail. It exhibits that the event loading time is the dominant contributor to the slowdown compared to the element tree's construction time and diffing time.



(a) change-based comparison time

(b) state-based comparison time



(c) change-based memory footprint

(d) state-based memory footprint

Figure 10 – Breakdown view of comparison time and memory footprint in Figure 9.

For the state-based comparison in Fig. 10b, the comparison time only experiences a slight increase as the number of identified differences also grows. This slight increase is contributed mainly by the diffing time, while the matching time tends to be constant due to the very small increase of the total elements (Figures 8).

Nevertheless, change-based comparison generally consumes more memory than the state-based comparison (see Figure 9b). It only consumes less memory than its state-based counterpart when the number of events is less than 0.3 million (around less than 0.25 million identified differences at that moment). Fig. 10c breaks down the memory footprint of change-based comparison into three factors: the loaded change events, element tree, and diffs. As modification continues, an increasing number of events is generated. These events have to be loaded into memory since they contain

the required information for the construction of an element tree. The amount of space to keep these change events in memory grows linearly with their number.

In contrast, the memory used for the element tree grows logarithmically. As the number of events increases, the probability that events modify already affected elements also increases. Thus, no additional memory allocation is required for the element tree. We can also notice that the element tree occupies most of the memory footprint since it mirrors the partial states – elements, features, and values – of the models that are affected by the changes. Moreover, in our technical implementation, a feature can have many instances – one instance for each element (As a comparison, in the EMF implementation, there is only one instance for a feature. The feature is used as a key so that different elements can have the same feature that maps to different values simultaneously). This contributes to the large memory footprint used by the element tree. The identified change-based diffs, the third factor, are the smallest factor that contributes to the memory footprint of the change-based comparison.

For the state-based comparison in Fig. 10d, the memory footprint only grows slightly along the increase of differences. A large part of the memory footprint is used to represent the identified differences, while the memory used for matches tends to be constant as the changes of the total elements are very small – less new elements means less memory needs to be allocated for new matches (Figures 8).

5.2.2 Homogeneous Operations

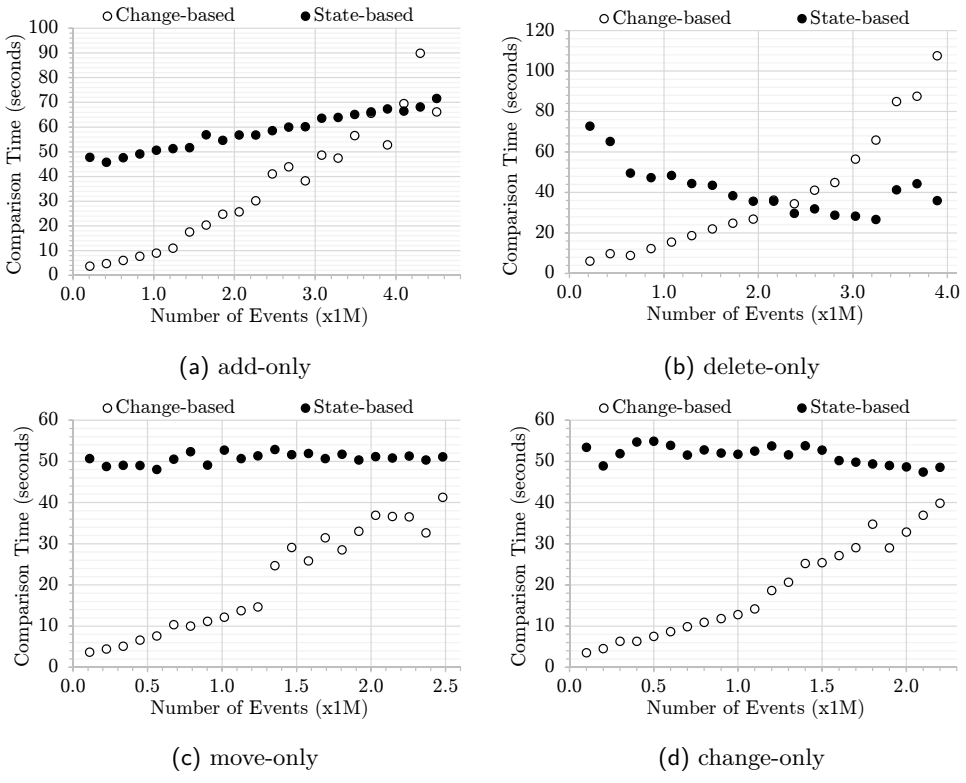


Figure 11 – Comparison time for homogeneous operations.

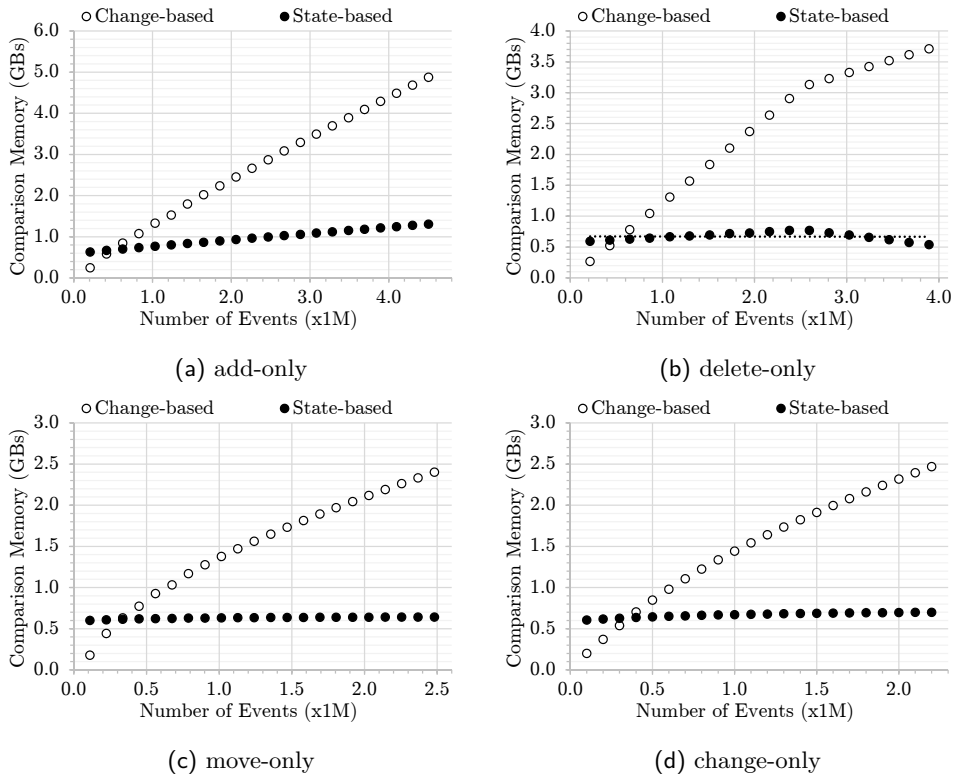


Figure 12 – Memory footprint for homogeneous operations.

Figures 11 and 12 exhibit the comparison time and memory footprint of models that have been modified using homogeneous operations – add, remove, move, or set only. We can notice that in all figures change-based comparison outperforms its state-based counterpart, particularly when the number of change events is small relative to the size of the model. As the number of modifications grows, eventually change-based comparison becomes slower than state-based comparison. In our experiments, this happens when the number of events is greater than 4 million (Fig. 11a). Change-based comparison also becomes slower when the size of models shrinks (due to a large number of delete events) as depicted in Fig. 12b as the change-based comparison still needs to load these change events and construct its element tree; in contrast, deletion means less work for state-based comparison. In terms of memory footprint, change-based comparison only performs better than state-based comparison when the number of change events is less than 0.3 millions as depicted in Fig. 12.

Based on the findings, we argue that the change-based comparison approach works at its best for large models that have been modified a moderate number of times. Models that have been excessively modified and experience significant reduction on model size could impair the performance of change-based comparison as a great number of change records have to be read and loaded into memory.

5.3 Limitations and Validity

The evaluation of the proposed change-based comparison is limited to the Java metamodel only. Thus, there is no guarantee it will perform in a consistent manner on models conforming to different metamodels. Although, we have tried to cover as much as common changes made in EMF models (e.g., performing add/remove/set/move operations on single/multi-valued features, attribute/reference features, or containment/non-containment references), the random modification made in the evaluation does not largely reflect the evolution of models in the real world. This is challenging as different domains can have their own patterns of model evolution – different problems, metamodels, modellers, etc.

6 Related Work

We are not aware of any other work that targets comparison and diffing of change-based models persisted as files. However, there are several existing tools for state-based model comparison. Beyond EMFCompare, which we used for our comparative evaluation due to its maturity and ongoing development activity, tools such as SiDiff [TBWK07] and DSMDiff [LGJ07] also provide language-agnostic graph-based model comparison, with some room for configuration (e.g., assigning different weights to features of types in the language). Additional expressive power – at the cost of increased complexity and configuration effort – is offered by dedicated comparison languages such as the Epsilon Comparison Language, which can be used to compare both homogeneous and heterogeneous models [Kol09]. We refrain from a more detailed discussion on state-based comparison tools as they all require upfront loading of both versions of the model into memory, which is the main cost that we aspire to reduce with the presented change-based approach.

Database-backed model persistence and version control solutions such as CDO [Ecla] and EMFStore [KH10] also provide diffing capabilities between different versions of the same model without requiring models to be fully loaded into memory, however they present integration challenges with mainstream software engineering tools (e.g., continuous integration systems, backup and restore facilities) which are typically file-based, and their performance can degrade as more models/users are added to a repository, since all models are effectively stored in a single database [KRM+13].

7 Conclusions and Future Work

In this paper, we have presented a novel approach to model comparison by exploiting the nature of change-based persistence which allows us to find differences between versions of a model by only comparing the last set of changes between the source and reference model. Our evaluation results suggest that using this approach, we can produce model comparison that is faster than traditional, state-based model comparison. However, the change-based comparison approach needs to load change events from a change-based persistence into main memory and thus may requires more memory than for state-based comparison. In our evaluation, this occurs when the number of change events exceeds 400,000. Arguably, diff and merge operations are usually performed on smaller deltas than our evaluation. The next challenge for future

work is to identify strategies to merge models optimally and persist the merging in the change-based way.

References

[BKL+12] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. An introduction to model versioning. In *Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*, pages 336–398, 2012. URL: https://doi.org/10.1007/978-3-642-30982-3_10, doi:10.1007/978-3-642-30982-3_10.

[Ecla] Eclipse. Eclipse CDO The Model Repository. https://www.eclipse.org/cdo/documentation/. Accessed: 2019-04-02.

[Eclb] Eclipse. EMF Compare. https://www.eclipse.org/emf/compare/. Accessed: 2018-01-15.

[Eclc] Eclipse. Epsilon. https://www.eclipse.org/epsilon/. Accessed: 2018-02-12.

[Ecld] Eclipse. Epsilon Git. http://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git/commit/?id=ebd0991c279a1f0df1acb529367d2ace5254fe87. Accessed: 2018-02-19.

[Ecle] Eclipse. Java Metamodel. https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.modisco.java.doc%2Fmediawiki%2Fjava_metamodel%2Fuser.html. Accessed: 2019-01-08.

[EMF] EMFCompare. Emf compare developer guide. https://www.eclipse.org/emf/compare/documentation/latest/developer/developer-guide.html. Accessed: 2018-11-01.

[KH10] Maximilian Koegel and Jonas Helming. Emfstore: a model repository for EMF models. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 307–308, 2010. URL: http://doi.acm.org/10.1145/1810295.1810364, doi:10.1145/1810295.1810364.

[Kol09] Dimitrios S. Kolovos. Establishing correspondences between models with the epsilon comparison language. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *Model Driven Architecture - Foundations and Applications*, pages 146–157, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[KRM+13] Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos Matragkas, Richard F. Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan de Lara, István Ráth, Dániel Varró, Massimo Tisi, and Jordi Cabot.

A research roadmap towards achieving scalability in model driven engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering, Budapest, Hungary, June 17, 2013*, page 2, 2013.

[LGJ07] Yuehua Lin, Jeff Gray, and Frédéric Jouault. Dsmdiff: a differentiation tool for domain-specific models. *European Journal of Information Systems*, 16(4):349–361, 2007. URL: https://doi.org/10.1057/palgrave.ejis.3000685, arXiv:https://doi.org/10.1057/palgrave.ejis.3000685, doi:10.1057/palgrave.ejis.3000685.

[Mye86] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986. URL: https://doi.org/10.1007/BF01840446, doi:10.1007/BF01840446.

[SBMP08] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Eclipse Series. Pearson Education, 2008. URL: https://books.google.co.uk/books?id=sA0zOZuDXhgC.

[TBWK07] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. Difference computation of large models. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 295–304, New York, NY, USA, 2007. ACM. URL: http://doi.acm.org/10.1145/1287624.1287665, doi:10.1145/1287624.1287665.

[YKP17] Alfa Yohannis, Dimitris S. Kolovos, and Fiona Polack. Turning models inside out. In *Proceedings of MODELS 2017 Satellite Events co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, TX, USA, September, 17, 2017.*, pages 430–434, 2017. URL: http://ceur-ws.org/Vol-2019/flexmde_8.pdf.

[YRPK18a] Alfa Yohannis, Horacio Hoyos Rodriguez, Fiona Polack, and Dimitris S. Kolovos. Towards efficient loading of change-based models. In *Modelling Foundations and Applications - 14th European Conference, ECMFA 2018, Held as Part of STAF 2018, Toulouse, France, June 26-28, 2018, Proceedings*, pages 235–250, 2018. URL: https://doi.org/10.1007/978-3-319-92997-2_15, doi:10.1007/978-3-319-92997-2_15.

[YRPK18b] Alfa Yohannis, Horacio Hoyos Rodriguez, Fiona Polack, and Dimitris S. Kolovos. Towards hybrid model persistence. In *Proceedings of MODELS 2018 Workshops co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018.*, pages 594–603, 2018. URL: http://ceur-ws.org/Vol-2245/me_paper_3.pdf.

Acknowledgments This work was partly supported by through a scholarship managed by *Lembaga Pengelola Dana Pendidikan Indonesia* (Indonesia Endowment Fund for Education).